Bean Binding

Bean Binding

Bean Binding in Camel defines both which methods are invoked and also how the Message is converted into the parameters of the method when it is invoked.

Choosing the method to invoke

The binding of a Camel Message to a bean method call can occur in different ways, in the following order of importance:

- if the message contains the header **CamelBeanMethodName** then that method is invoked, converting the body to the type of the method's argument.
  - From **Camel 2.8** onwards you can qualify parameter types to select exactly which method to use among overloads with the same name (see below for more details).
  - From **Camel 2.9** onwards you can specify parameter values directly in the method option (see below for more details).
- you can explicitly specify the method name in the DSL or when using POJO Consuming or POJO Producing
- if the bean has a method marked with the `@Handler` annotation, then that method is selected
- if the bean can be converted to a Processor using the Type Converter mechanism, then this is used to process the message. The ActiveMQ component uses this mechanism to allow any JMS MessageListener to be invoked directly by Camel without having to write any integration glue code. You can use the same mechanism to integrate Camel into any other messaging/remoting frameworks.
- if the body of the message can be converted to a BeanInvocation (the default payload used by the ProxyHelper) component - then that is used to invoke the method and pass its arguments
- otherwise the type of the body is used to find a matching method; an error is thrown if a single method cannot be chosen unambiguously.
- you can also use Exchange as the parameter itself, but then the return type must be void.
- if the bean class is private (or package-private), interface methods will be preferred (from **Camel 2.9** onwards) since Camel can't invoke class methods on such beans

In cases where Camel cannot choose a method to invoke, an `AmbiguousMethodCallException` is thrown.

By default the return value is set on the outbound message body.

Asynchronous processing

From **Camel 2.18** onwards you can return a CompletionStage implementation (e.g. a CompletableFuture) to implement asynchronous processing.

Please be sure to properly complete the CompletionStage with the result or exception, including any timeout handling. Exchange processing would wait for completion and would not impose any timeouts automatically. It's extremely useful to monitor Inflight repository for any hanging messages.

Note that completing with "null" won't set outbody message body to null, but would keep message intact. This is useful to support methods that don't modify exchange and return CompletableFuture<Void>. To set body to null, just add Exchange method parameter and directly modify exchange messages.

Examples:

Simple asynchronous processor, modifying message body.

```
public CompletableFuture<String> doSomethingAsync(String body)
```

Composite processor that do not modify exchange

```
public CompletableFuture<Void> doSomethingAsync(String body) {
    return CompletableFuture.allOf(doA(body), doB(body), doC());
}
```

Parameter binding

When a method has been chosen for invocation, Camel will bind to the parameters of the method.

The following Camel-specific types are automatically bound:

- `org.apache.camel.Exchange`
- `org.apache.camel.Message`
- `org.apache.camel.CamelContext`
- `org.apache.camel.TypeConverter`
- `org.apache.camel.spi.Registry`
- `java.lang.Exception`

So, if you declare any of these types, they will be provided by Camel. **Note that `Exception` will bind to the caught exception of the Exchange** - so it's often usable if you employ a Pojo to handle, e.g., an `onException` route.

What is most interesting is that Camel will also try to bind the body of the Exchange to the first parameter of the method signature (albeit not of any of the types above). So if, for instance, we declare a parameter as `String body`, then Camel will bind the IN body to this type. Camel will also automatically convert to the type declared in the method signature.

Let's review some examples:

Below is a simple method with a body binding. Camel will bind the IN body to the `body` parameter and convert it to a `String`.

```
public String doSomething(String body)
```

In the following sample we got one of the automatically-bound types as well - for instance, a `Registry` that we can use to lookup beans.

```
public String doSomething(String body, Registry registry)
```

We can use Exchange as well:

```
public String doSomething(String body, Exchange exchange)
```

You can also have multiple types:

```
public String doSomething(String body, Exchange exchange, TypeConverter converter)
```

And imagine you use a Pojo to handle a given custom exception `InvalidOrderException` - we can then bind that as well:

```
public String badOrder(String body, InvalidOrderException invalid)
```

Notice that we can bind to it even if we use a sub type of `java.lang.Exception` as Camel still knows it's an exception and can bind the cause (if any exists).

So what about headers and other stuff? Well now it gets a bit tricky - so we can use annotations to help us, or specify the binding in the method name option.
See the following sections for more detail.

Binding Annotations

You can use the Parameter Binding Annotations to customize how parameter values are created from the Message

Examples

For example, a Bean such as:

```
public class Bar {
    public String doSomething(String body) {
    // process the in body and return whatever you want
    return "Bye World";
}
```

Or the Exchange example. Notice that the return type must be **void** when there is only a single parameter of the type `org.apache.camel.Exchange`:

```
 public class Bar {
     public void doSomething(Exchange exchange) {
         // process the exchange
         exchange.getIn().setBody("Bye World");
 }
```

@Handler

You can mark a method in your bean with the @Handler annotation to indicate that this method should be used for Bean Binding.
This has an advantage as you need not specify a method name in the Camel route, and therefore do not run into problems after renaming the method in an IDE that can't find all its references.

```
public class Bar {
    @Handler
    public String doSomething(String body) {
        // process the in body and return whatever you want
        return "Bye World";
    }
}
```

Parameter binding using method option

Available as of Camel 2.9

Camel uses the following rules to determine if it's a parameter value in the method option

- The value is either `true` or `false` which denotes a boolean value
- The value is a numeric value such as `123` or `7`
- The value is a String enclosed with either single or double quotes
- The value is null which denotes a `null` value
- It can be evaluated using the Simple language, which means you can use, e.g., body, header.foo and other Simple tokens. Notice the tokens must be enclosed with ${ }.

Any other value is consider to be a type declaration instead - see the next section about specifying types for overloaded methods.

When invoking a Bean you can instruct Camel to invoke a specific method by providing the method name:

```
.bean(OrderService.class, "doSomething")
```

Here we tell Camel to invoke the doSomething method - Camel handles the parameters' binding. Now suppose the method has 2 parameters, and the 2nd parameter is a boolean where we want to pass in a true value:

```
public void doSomething(String payload, boolean highPriority) {
    ...
}
```

This is now possible in **Camel 2.9** onwards:

```
.bean(OrderService.class, "doSomething(*, true)")
```

In the example above, we defined the first parameter using the wild card symbol *, which tells Camel to bind this parameter to any type, and let Camel figure this out. The 2nd parameter has a fixed value of `true`. Instead of the wildcard symbol we can instruct Camel to use the message body as shown:

```
.bean(OrderService.class, "doSomething(${body}, true)")
```

The syntax of the parameters is using the Simple expression language so we have to use ${ } placeholders in the body to refer to the message body.

If you want to pass in a `null` value, then you can explicit define this in the method option as shown below:

```
.to("bean:orderService?method=doSomething(null, true)")
```

Specifying `null` as a parameter value instructs Camel to force passing a `null` value.

Besides the message body, you can pass in the message headers as a `java.util.Map`:

```
.bean(OrderService.class, "doSomethingWithHeaders(${body}, ${headers})")
```

You can also pass in other fixed values besides booleans. For example, you can pass in a String and an integer:

```
.bean(MyBean.class, "echo('World', 5)")
```

In the example above, we invoke the echo method with two parameters. The first has the content 'World' (without quotes), and the 2nd has the value of 5. Camel will automatically convert these values to the parameters' types.

Having the power of the Simple language allows us to bind to message headers and other values such as:

```
.bean(OrderService.class, "doSomething(${body}, ${header.high})")
```

You can also use the OGNL support of the Simple expression language. Now suppose the message body is an object which has a method named `asXml`. To invoke the `asXml` method we can do as follows:

```
.bean(OrderService.class, "doSomething(${body.asXml}, ${header.high})")
```

Instead of using `.bean` as shown in the examples above, you may want to use `.to` instead as shown:

```
.to("bean:orderService?method=doSomething(${body.asXml}, ${header.high})")
```

Using type qualifiers to select among overloaded methods

Available as of Camel 2.8

If you have a Bean with overloaded methods, you can now specify parameter types in the method name so Camel can match the method you intend to use. Given the following bean:

```
 from("direct:start")
    .bean(MyBean.class, "hello(String)")
    .to("mock:result");
```

Then the `MyBean` has 2 overloaded methods with the names `hello` and `times`. So if we want to use the method which has 2 parameters we can do as follows in the Camel route:

```
from("direct:start")
    .bean(MyBean.class, "hello(String,String)")
    .to("mock:result");
```

We can also use a * as wildcard so we can just say we want to execute the method with 2 parameters we do

```
 from("direct:start")
    .bean(MyBean.class, "hello(*,*)")
    .to("mock:result");
```

By default Camel will match the type name using the simple name, e.g. any leading package name will be disregarded. However if you want to match using the FQN, then specify the FQN type and Camel will leverage that. So if you have a `com.foo.MyOrder` and you want to match against the FQN, and **not** the simple name "MyOrder", then follow this example:

```
.bean(OrderService.class, "doSomething(com.foo.MyOrder)")
```

Camel currently only supports either specifying parameter binding or type per parameter in the method name option. You **cannot** specify both at the same time, such as

```
doSomething(com.foo.MyOrder ${body}, boolean ${header.high})
```

This may change in the future.